0-1 knapsack

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
// Function to solve 0-1 Knapsack using dynamic programming
int knapsack(int W, vector<int> &wt, vector<int> &val, int n) {
   // Create a 2D table to store results of subproblems
   vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));
   // Build the dp array bottom-up
   for (int i = 1; i <= n; i++) {
      for (int w = 1; w <= W; w++) {
         if (wt[i - 1] <= w) {
            dp[i][w] = max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1][w]);
         } else {
            dp[i][w] = dp[i - 1][w];
         }
      }
   }
   // Return the maximum value that can be stored in the knapsack of capacity W
   return dp[n][W];
}
int main() {
   int W = 50;  // Knapsack capacity
   vector<int> wt = {10, 20, 30};  // Weights of items
   vector<int> val = {60, 100, 120};  // Values of items
   int n = wt.size();  // Number of items
   int maxValue = knapsack(W, wt, val, n);
   cout << "Maximum value in Knapsack = " << maxValue << endl;

   return 0;}
```

// Write a program non-recursive and recursive program to calculate Fibonacci numbers and

// analyze their time and space complexity.

```
# include <iostream>
using namespace std;

int non_recursive(int n){
   if(n<=1){
      return n;
   }

   int prev = 0;
   int curr = 1;
   int next;
   for(int i=2;i<=n;i++){
      next = curr+prev;
      prev = curr;
      curr = next;
   }
   return next;

}

int recursive(int n){
   if(n <=1){
      return n;
   }
   int ans = recursive(n-1)+recursive(n-2);
   return ans;
}

int main(){
```

```
    int n = 5;
    int ans = non_recursive(n);
    cout<<ans;
}

// fractional Knapsack
// take profit and weight as input ;
// store them in a vector
// sort the vector accroding to profit and weight ratio
//  add elment in knapsack

#include<iostream>
#include <vector>
#include<algorithm>
using namespace std;
class Item{
    public:

        int weight;
        int profit;

        Item(int weight,int profit){
            this->profit = profit;
            this->weight = weight;
        }
};

bool compare(Item a,Item b){
    int r1 = a.profit/a.weight;
    int r2 = b.profit/b.weight;
```

```
    return (r1 > r2);

}

int main(){

    char response ='y';
    vector<Item>store;
    cout<<"enter capacity of knapsack ";
    int capacity;
    cin>>capacity;
    int temp = capacity;
    do{
        int w;
        int p;
        cout<<"enter weight : "<<endl;
        cin>>w;
        cout<<"enter profit : "<<endl;
        cin>>p;
        cout<<endl<<endl;

        cout<<"do you want to continue ?(y / n)";
        cin>>response;
        Item i =  Item(w,p);
        store.push_back(i);

    }
    while(response == 'y');

    // sort the vector;
```

```
    sort(store.begin(),store.end(),compare); // O(nlogn);

    // enter item into knapsack
    int index = 0;
    int total = 0;
    while(capacity >0){ // O(n);

        Item current = store[index];
        if(current.weight <= capacity){
            total+=current.profit;
            capacity = capacity-current.weight;
            cout<<"capacity :"<<capacity<<endl;
        }
        else {

            double fraction = ((double)capacity/current.weight)*current.profit;
            cout<<"fraction is : "<<fraction<<endl;
            total+=fraction;
            capacity = 0;
        }
        index++;
    }

    cout<<"----------+--------------+----------------+--------------+------------"<<endl;
    cout<<"capacity : "<<temp<<endl;
    cout<<"max profit is : "<<total;
};
```

```
//HUFFMAN ENCODING
//count frequencies of each character
// and store it in a map;
// insert all the character along with their frequencies in minheap
// build a huffman tree form the minheap;
// build huffman code form the huffman tree;

#include<iostream>
#include<map>
#include<queue>
using namespace std;

class Node{
   public:
   char ch;
   int freq;
   Node * left;
   Node * right;


   Node(char ch,int freq){
      this->ch = ch;
      this->freq = freq;
      left = NULL;
      right = NULL;
   }
};

// since we are storing character and its frequencies together
// we need to define custom comparator for priority_queue;
// for that declare class Compare and overload operator() method;
```

```
class Compare {
   public:
     bool operator()(Node * a, Node* b){
        if(a->freq > b->freq){
           return true;
        }
        return false;
    }
};


// now build huffmantree;
   Node * build( priority_queue<Node*,vector<Node*>,Compare>pq){
      while(pq.size()>1){
         Node * first = pq.top();
         pq.pop();
         Node * second = pq.top();
         pq.pop();
         int s = first->freq+second->freq;
         Node*sum = new Node(NULL,s);
         sum->left = first;
         sum->right = second;
         pq.push(sum);

      }

      Node * root = pq.top();
      return root;

   }
```

```
// now buildcode
// for traverse the entire tree in dfs manner
// while traversing build the code
// 0 for left and 1 for right
// if leaf node encountered then store the sequence;

void buildcode(Node*root,map<char,string>&mp,string code){
    if(root == NULL){
        return;
    }
    // if leaf node
    if(root->left == NULL && root->right==NULL){
        mp[root->ch] = code;
    }
    else{
        if(root->left !=NULL){
        buildcode(root->left,mp,code+"0");
            }
        if(root->right !=NULL){
        buildcode(root->right,mp,code+"1");
            }
}
}




int main(){
    string str;
    cout<<"enter string : ";
    cin>>str;
```

```
    map<char,int>mp;
    for(int i=0;i<str.length();i++){
        mp[str[i]]++;
    }
    priority_queue<Node*,vector<Node*>,Compare>pq;
    // here we have inserted node into pririty queue;
    for(auto i :mp){
        Node* temp = new Node(i.first,i.second);
        pq.push(temp);
    }
    Node *root =  build(pq);
    map<char,string>mn;
    string code = "";
    buildcode(root,mn,code);
    for(auto i :mn){
        cout<<i.first <<" "<<i.second<<endl;

    }
}
```

// NQUEEN

```
#include <iostream>
#include <vector>

using namespace std;

// Utility function to print the board
void printBoard(vector<vector<int>> &board, int N) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            cout << (board[i][j] == 1 ? "Q " : ". ");
        }
        cout << endl;
    }
}

// Function to check if a queen can be placed at board[row][col]
bool isSafe(vector<vector<int>> &board, int row, int col, int N) {
    // Check the left side of the row
    for (int i = 0; i < col; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }

    // Check upper diagonal on the left side
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }
```

```
    // Check lower diagonal on the left side
    for (int i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }

    return true;
}

// Recursive function to solve N-Queens problem
bool solveNQueensUtil(vector<vector<int>> &board, int col, int N) {
    // If all queens are placed, return true
    if (col >= N) {
        return true;
    }

    // Try placing the queen in all rows for this column
    for (int i = 0; i < N; i++) {
        if (isSafe(board, i, col, N)) {
            board[i][col] = 1;  // Place the queen

            // Recur to place the rest of the queens
            if (solveNQueensUtil(board, col + 1, N)) {
                return true;
            }

            // If placing queen at board[i][col] doesn't lead to a solution,
            // then backtrack by removing the queen
            board[i][col] = 0;
```

```
        }
    }

    // If the queen cannot be placed in any row in this column, return false
    return false;
}

// Function to solve the N-Queens problem with the first queen already placed
bool solveNQueens(int N, int firstRow, int firstCol) {
    vector<vector<int>> board(N, vector<int>(N, 0));

    // Place the first queen at the specified position
    board[firstRow][firstCol] = 1;

    // Start placing queens from the next column
    if (!solveNQueensUtil(board, firstCol + 1, N)) {
        cout << "No solution exists" << endl;
        return false;
    }

    // Print the board if a solution is found
    printBoard(board, N);
    return true;
}

int main() {
    int N = 8;          // Size of the board (NxN)
    int firstRow = 0;   // Row where the first queen is placed
    int firstCol = 0;   // Column where the first queen is placed

    if (!solveNQueens(N, firstRow, firstCol)) {
```

```
        cout << "No solution exists for the given initial queen placement." << endl;
    }

    return 0;
}
```